Fundamental Algorithm Homework 5 Solution

By Xing Xia

1.

(i) Give a simple algorithm for the grouping problem

Solution:

The algorithm below outputs a list $[n_1, n_2, \ldots n_k]$ that represents the subdivision as described in the problem.

```
Subdivision = [];
Sum = 0;
For i = 1 to n do {
        If  (Sum+W(i) <= M) {
                Sum += W(i);
        } else {
                Subdivision = Subdivision + [i-1];
                Sum = 0;
        }
}
Subdivision = Subdivision + [n];
Print (Subdivision);
```

(ii) Prove that your algorithm is optimal

Solution:

Let $[n_1, n_2, \ldots n_p]$ be the subdivision given by the above algorithm. Suppose it's not the optimal subdivision, then there exists a number q, where q < p, such that $[m_1, m_2, \ldots m_q]$ is also a possible solution to grouping problem.

Now, we want to show $n_k \geq m_k$, for any k, where $1 \leq k \leq p$, by induction.

It's easy to see $n_1 \geq m_1$, since in greedy algorithm in (i), we always try to make Sum as large as possible, as far as Sum+W(i) <= M.

Suppose when k = h, where h <p, n_k and m_k are both defined, $n_k \geq m_k$. Then for k = h+1, we claim if n_{h+1} is defined, so is m_{h+1}, and $n_{h+1} \geq m_{h+1}$.

Since the last element of any subdivision equals n, the size of the input list, and $n \geq n_{h+1} > n_h \geq m_h$, we know m_{h+1} must be defined.

Suppose $n_{h+1} < m_{h+1}$. Let

$G^n_h = (W(n_h), W(n_h + 1), \ldots W(n_{h+1}))$ and

$G^m_h = (W(m_h), W(m_h + 1), \ldots W(m_{h+1}))$

We know $size(G^n_h) \leq M$ and $size(G^m_h) \leq M$. Now, let's try to extend G^n_h as G^n_h', where $G^n_h' = (W(n_h), W(n_h + 1), \ldots W(n_{h+1}), W(n_{h+1}+1))$

According to the assumption, $n_h \geq m_h$ while $n_{h+1} < m_{h+1}$, we know

$size(G^n_h') \leq size(G^m_h) \leq M$

which contradicts with the greedy algorithm in (i), which implies G^n_h cannot be extended any longer. So for k = h+1, we have $n_k \geq m_k$

In particular, we have $n_p \geq m_p$, which implies $p \leq q$.

(iii) Suppose W(i) may be negative as well. Either prove that your algorithm is still optimal or show a counter example.

Solution:

The greedy algorithm in (i) is not optimal in this situation. A counter example:

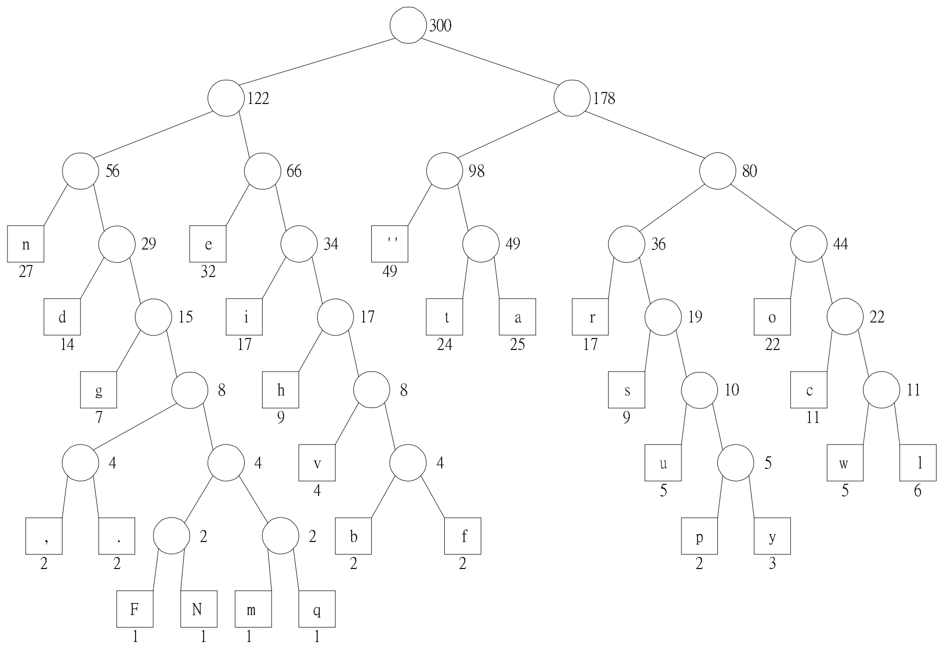
M = 5, w = (2, 4, -6).

The algorithm in (i) will give the solution [1, 3] , while the optimal solution is [3].

2.

(i) The overall bit length of the coded string is 1223 bits.

(ii)



3.

(1) Prove that a Huffman tree with n leaves has exactly 2n – 2 edges.

Solution:

Use induction of the structure of the tree.

For n = 1, 2n-2 = 0, there is only one node with no edges.

Assume when n = k, any Huffman tree with k leaves has 2k – 2 edges.

Let's see how many edges a Huffman tree with n = k + 1 leaves has.

Since Huffman tree is full binary tree, if we want to add one more leaf to a Huffman tree with n = k leaves, we have to replace one leaf with an internal node and 2 leaves. It means we add two more edges to the Huffman tree, i.e., the Huffman tree with k+1 have (2k – 2) + 2 = 2(k+1) – 2 edges.

So we claim that the Huffman tree with n leaves has exactly 2n – 2 edges.

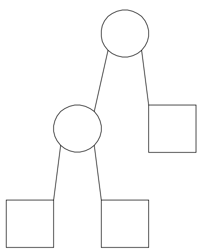
(2) Tell us how to construct the representation from any Huffman tree for set C = {0, …n-1}.

Solution:

We represent the Huffman tree by depth-first traversing the edges of the tree, using 0 for walking down the edge, using 1 for walking up the edge. Since the Huffman tree with n leaves has 2n-2 edges, and each edge is traversed twice (once walking up, once walking down), we use (2n - 2) * 2 = 4n-4 bits to represent the Huffman tree.

Now we associate the elements of C to the leaves of the Huffman tree. For each element in C, we need exactly [lgn] bits to represent it, thus n[lgn] in total for n elements in C. Another point is we need to represent the elements in the order of depth-first traverse. So the overall representation should use at most 4n – 4 + n[lgn] bits.

Alternative solution: we can improve the above solution by using 2n-1 bits instead of using 4n-4 bits.
In this method, we record the nodes (both internal nodes and leaves) of the Huffman tree row by row, starting from the root of the tree. If the node has two children, we use 1 to represent it; Otherwise, i.e. the node is a leaf, we use 0 to represent it.
For example, the representation of the Huffman tree below is 11000.



Since the Huffman tree with n leaves has 2n-1 nodes in total (you can prove it by induction of the structure of the tree), we need exactly 2n-1 nodes to represent the Huffman tree.
It's important to note that you can know when the bit string has reached the end of its description of a Huffman tree.

(3) Apply your code to the tree in Figure 16.4(b), assuming that a = 0, … f = 5.
Solution:
We could represent the Huffman tree in Figure 16.4(b) using the following binary code:
010001011000101101111 000 010 001 101 100 011
^^^^^^^^^^^^^^^^^^^^^ ^^^ ^^^ ^^^ ^^^ ^^^ ^^^
The representation of tree   a   c   b   f   e   d

(4) Describe how to reconstruct the Huffman tree from your representation.
Solution:
Note that we do not know in advance what n is and thus the separation between the first 4n-4 bits and the rest.
Look at the bits one by one, we use the following method to reconstruct the Huffman tree:
First, we create a root node with no children, and set it as the current node. We use root_visited_times to record the number of times the root node is visited. We initialize root_visited_times = 0.

1. If the current bit is 0 and the current node has no left child, create a left child for the current node and traverse to the left child.
2. If the current bit is 0 and the current node has left child, create a right child for the current node and traverse to the right child.
3. If the current bit is 1, traverse to the parent of current node. If the parent is root, root_visited_times ++. The process is finished when root_visited_times = 2;

We can do the same for the alternative 2n-1 bits solution.

4. The generalization of incrementing binary counters

Solution:

First we define

len(C): the length of bits of the counter C,

one(C): the number of bit "1" of the counter C.

$\Phi = \Sigma(3\text{len}(C) + \text{one}(C))$, for all counter C in collection

Now let's analyze the cost of operation add(C, C'). For simplicity, we only consider the situation when len(C) >= len(C').

As the hint given by professor Yap, the cost of add(C, C') is the number of bits of C and C' that you need to look at. We define d be the difference of the number of bits of C and C' you need to look at. For example, if C = 10011101 and C' = 110, then C+C' = 10100011. The cost is 9, this is because you need to look at 6 bits of C and 3 bits of C'. On the other hand, d = 6 – 3 = 3, as the above definition of d. We can compute the cost with the following equation:

Cost = 2len(C') + d

Now, let's estimate $\Delta\Phi$, the increase in potential during the operation add(C, C'). Since C' is set to zero after the add(C, C') operation, $\Delta\Phi$ is charged by – 3len(C') - one(C'). On the other hand, Len(C) is increased at most by 1.

Let's see how one(C) changed after the add operation. For the lowest len(C') bits of the counter C, the increased number of bit "1" is at most len(C'). For the next d bits of the counter C, the decreased number of bit "1" is exactly d-1. So, in total,

$\Delta\Phi$ = -3len(C') – one(C') + 3*1 + len(C') – (d-1) <= -2len(C') – d + 4

The amortized cost for add(C, C') operation therefore

Cost' = Cost + $\Delta\Phi$ <= (2len(C') + d) + (-2len(C') – d + 4) = 4.

Thus the amortized cost for add(C, C') is a constant.

As we know from the textbook, the amortized cost for inc(C) is 2, a constant, too.

So we know the problem has an amortized cost that is constant per operation.

5. Problem 17-2, page 426

(a) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

Solution:

Search k sorted arrays $A_0, A_1, \ldots, A_{k-1}$ one by one until the searched element is found; for each array A_i (0 <= I < k), do binary search.

The worst-case running time $T(n) = 1 + 2 + \ldots + k = k(k+1)/2 = O(k^2) = O(\lg^2 n)$

(b) Describe how to insert a new element into this data structure. Analyze its worst-case and amortized running times.

Solution:

How to insert a new element?

Check $A_0, A_1, \ldots, A_{k-1}$ in order one by one, find the first empty array A_i.

If i = 0, insert the new element in A_0 directly. Otherwise, sort all the elements in $A_0, \ldots A_{i-1}$ and the inserted new element, put these sorted elements in Ai and empty $A_0, \ldots A_{i-1}$.

Worst case running time:

The worst case occurs when all $A_0, \ldots A_{k-2}$ are full and A_{k-1} is empty before the insertion.

We need to sort all the elements in A_0, … A_{k-2} and the new inserted elements and insert them into A_{k-1}. We can merge these arrays one by one, the total running time is O(n).

Amortized running time:
Using the aggregate analysis, we observe that the cost to fill A_i is 2^i, and A_i is filled after every 2^{i+1} insertions (Actually after every 2^i insertions, A_i is filled, after another 2^i insertions A_i is emptied, and so on). It means Ai will be filled $[n/2^{i+1}]$ times in total. So the total cost for N insertions is

$$\sum_{i=0}^{k}(2^i\left\lfloor\frac{n}{2^{i+1}}\right\rfloor) \le n\sum_{i=0}^{k}\frac{1}{2} = \frac{1}{2}n\sum_{i=0}^{k}1 = \frac{1}{2}n(k+1) = \frac{1}{2}n(\lceil\lg(n+1)\rceil).$$

Hence the amortized cost per operation is $\frac{1}{2}n(\lceil\lg(n+1)\rceil)/n = \frac{1}{2}(\lceil\lg(n+1)\rceil) = O(\lg n)$

(c) Discuss how to implement DELETE
Solution:
When deleting an element, say X, first perform search to locate the X.
If X is in A_0, delete it directly from A_0. Otherwise, we exchange the element X with an element Y in A_j, where A_j is the first non-empty array starting from A_0. If $j=0$, just empty A_0. Else, delete the elements in A_j and break A_j into $A_0 \sim A_{j-1}$. Keep the order in A_i and A_j.